Parallelization of K-Means Clustering Algorithm

Gopinath Dayalan

1. Abstract

K-means clustering is one of the popular cluster analysis technique in data mining. It is a technique of grouping data points with similar features and continuously assigns membership to every new data point encountered. The clustering is computationally difficult, it is a NP-hard problem. The efficient heuristic algorithm helps the iteration converge faster to the local optimum. In this project, the computation of k-means clustering is parallelized using p-thread and OpenMP. This is achieved by parallelizing membership assignment and calculation of the sum of squared errors (SSE). Multithreading is a popular way of improving computation performance. Thread creation, context switching, and communication between threads are much faster in threads and these threads can be terminated easily.

2. Introduction

K-means aims to partition N dataset into K subset based on the similarity where each cluster has a minimum nearest neighbor mean. K-means is often confused with the k-nearest neighbor (KNN) where KNN decides the ground truth based on its nearest neighbor's ground truth. Message Passing Interface (MPI), POSIX-Threads(P-Threads), OpenMP and CUDA are some of the parallelization techniques, from which P-thread and OpenMP are chosen for this project and the results between these are evaluated.

A thread is a single sequence or flow of work inside a process, they are also called a lightweight process. P-thread is independent parallel execution model which controls multiple different flows of work at a time. Every flow of work is called a thread, the creation and management of these threads are handled by P-thread API. All procedure calls to P-thread are prefixed with "pthread_" and they are of four groups, Thread Management which handles creation and joining of thread, mutex, condition variables, and synchronization.

OpenMP (Open-Multi Processing) is also a parallelization technique which supports multi-platform shared memory programming. OpenMP provides programmers a simple interface to develop parallel applications. Most of the variables are visible to all the threads in the program, but some needs to be kept private to avoid race conditions and at times some data needs to be communicated between threads. These types of conditions are handled by OpenMP directives, the different types of clauses are Data sharing attribute clauses, Synchronization clauses, Scheduling clauses, IF control, Initialization, Data copying, and Reduction.

3. Problem Description

K-Means algorithm takes the number of clusters and dataset as the input. The algorithm can be executed in two different approaches, where the number of clusters is predefined or iterated through a range of k values to figure out the optimum value for k. The latter approach is used in this project where the dataset iterated for 'k' in the range of 1 to 10. The optimal value of k is decided based on the sum of squared errors (SSE). Running through the dataset which has thousands of data points and figuring out the membership, calculating the new centroid and computing the errors is a time-consuming operation. The same operation must be carried out until the centroids remain unchanged. Hence, to increase the computational speed along with the high value of k, a parallel algorithm is required.

4. Algorithm

- **Serial Algorithm**

i. Get the k range.
ii. Randomly choose k number of datapoints as initial centroids.
iii. For each datapoint find the nearest centroid. Use Euclidean distance to find the nearest and assign membership.

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2$$

Where x is the dataset (x_1, x_2, x_3, … x_n) and μ is the centroids (μ_1, μ_2, μ_3, … μ_n)

iv. Calculate the new centroids for each k clusters by taking mean.

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j$$

v. Repeat until the centroids doesn't change or the change is below threshold.
vi. Calculate Sum of squared errors for evaluating the k value.
vii. Output k centroids and membership of each datapoints.
viii. Repeat for the next value of k

Complexity: O (n * K * I * d)

n = number of points	*~ 10,000*
K = number of clusters	*Let's say 10*
I = number of iterations	*Assume it's 10*
d = number of attributes	*dimension = 12*

Approx. ~ 12 Million calculations

- **Parallel Algorithm**

i. Get the k range
ii. Node 0 randomly choose k number of datapoints as initial centroids.
iii. Each thread is assigned with N/Threads datapoints.
iv. Thread calculates the membership for each point.
v. Join threads to calculate the new centroids.
vi. Repeat until the centroids doesn't change or the change is below threshold
vii. Assign N/threads data to each thread for SSE computation.
viii. Join SSE result from each thread for evaluation.
ix. Output k centroids and membership of each datapoints.
x. Repeat for the next value of k.

Critical Section:

P-Thread: Handled with mutex
OpenMP: Handled with reduction

Complexity: O ((n/thread count) * K * I * d)

n = number of points	*~ 10,000/10*
K = number of clusters	*Let's say 10*
I = number of iterations	*Assume it's 10*
d = number of attributes	*dimension = 12*

Approx. ~ 1.2 Million calculations (each thread)

5. Dataset

Credit card transaction dataset is used for k-means clustering. K means on this type of dataset help discover meaningful insight on the similarity of credit card usage and helps marketing new products based on the type of customers in a bank.

6. Evaluation

The code is executed with varying number of datasets i.e. 1000x12, 10000x12 and 100000x12. Average of 5 test observation is taken for serial, P-Thread and OpenMP code. The time taken to load the data is ignored in evaluation.

7. Result

When a range (1-10) value is chosen for k, the value for Sum of squared errors keeps reducing. It starts with ~4900 and goes till ~2800.

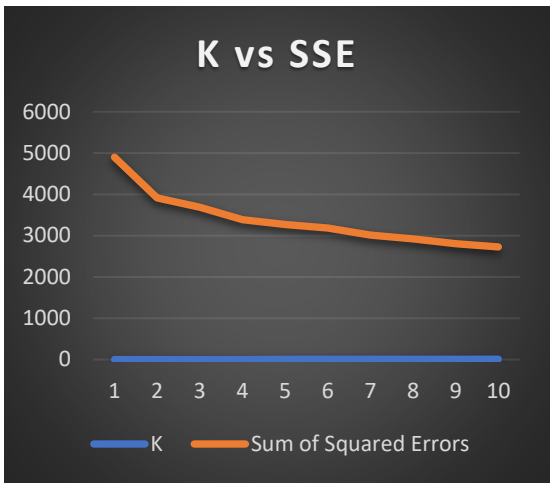

Figure 1) Comparison k value vs Sum of squared errors with serial code

In this range the optimal value for k is 10, but for higher value of k we end up in having duplicate centroids, it is better to have 'k' where SSE is low and there is no duplicate centroids

Time Elapsed

- **Serial Code**

		(in seconds)
No. of Datapoints	1000	0.358343
	10000	6.105595
	100000	47.315090

The increase in time is directly proportional to the increase in datapoints and it is exponential.

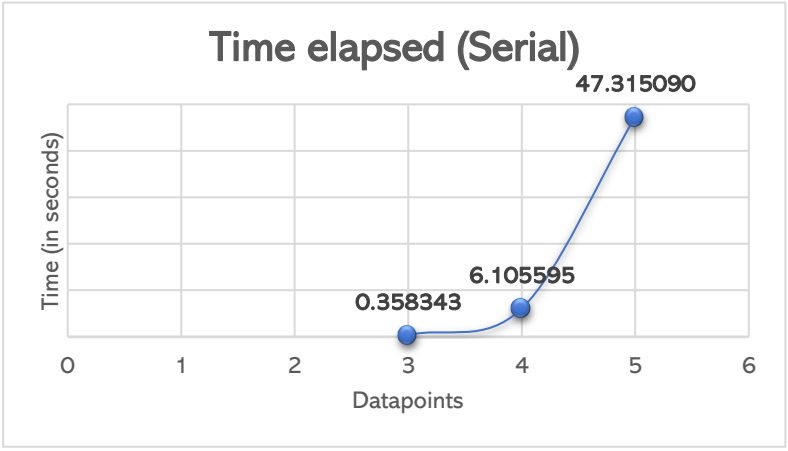

Figure 2) Datapoints (log base 10) vs time taken for serial code

- **P-Thread**

		No. of Threads			
		2	4	8	10
No. of Datapoints	1000	0.362271	0.235984	0.217588	0.214803
	10000	5.302395	3.935083	3.267711	2.857791
	100000	22.797150	16.971880	11.994180	11.788930

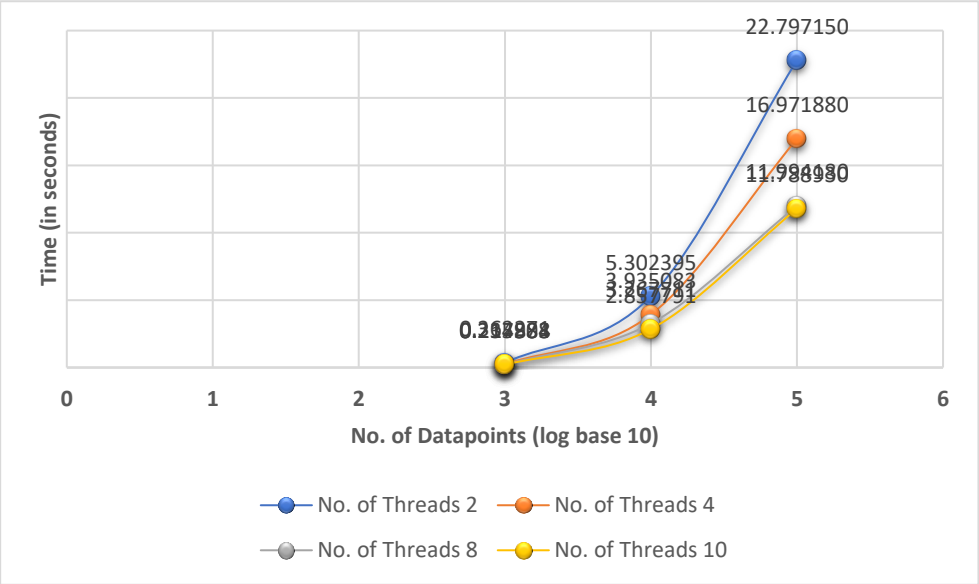

Figure 3) Datapoints (log base 10) vs time taken for p-thread code

When the algorithm is parallelized with p-thread there is a drastic reduction in the time elapsed with higher number of datapoints. The change in the time difference is minimal with 1000 data points. The increase in the number of threads improves the performance initially. From 8-10 thread there is only a slight increase in the performance.

- **OpenMP**

		No. of Threads			
		2	4	8	10
No. of Datapoints	1000	0.231435	0.189158	0.184874	0.173551
	10000	1.930362	1.569147	1.667151	1.646268
	100000	21.277540	15.975370	15.947590	16.425690

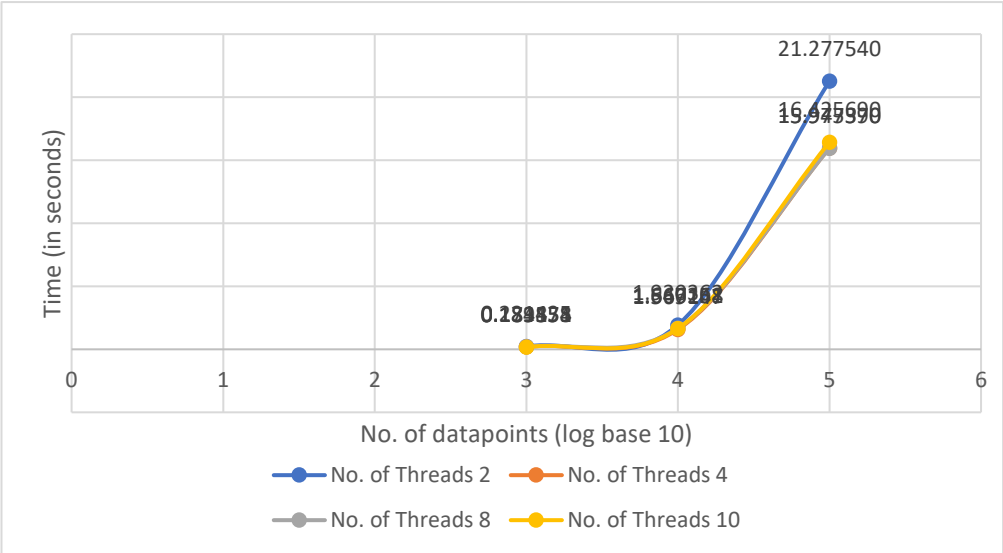

Figure 4) Datapoints (log base 10) vs time taken for OpenMP code

Like p-thread OpenMP also delivers better performance when there is thread count 2-8 and has an inverse effect while using 10 thread. This could be due to the performance overhead of content switching.

- **Speedup (Serial vs Parallel)**

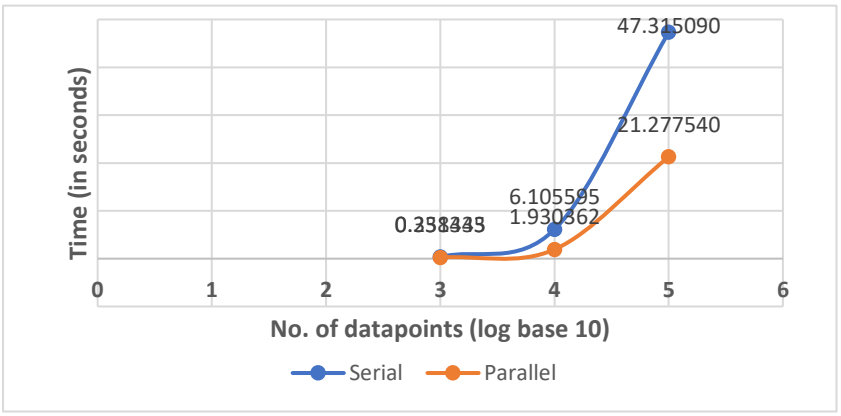

Figure 5) Datapoints (log base 10) vs time taken for OpenMP code

Parallel algorithm performs way better than serial algorithm where the execution time has been brought down to half. Considering the dataset with 1000 datapoints and 100000 datapoints, performance improvement is achieved with parallelization

- **P-Thread vs OpenMP comparison**

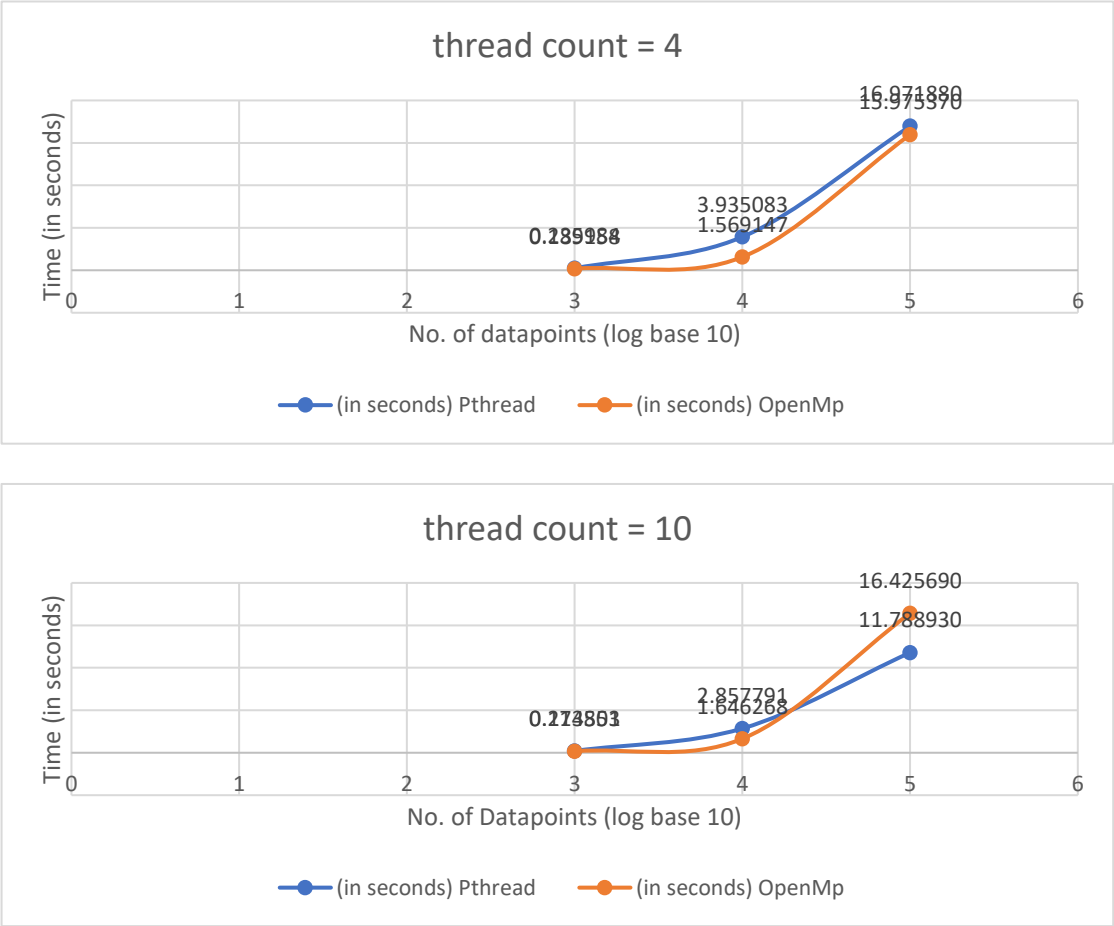

Figure 6) Comparison of performance between P-Thread and OpenMP with different thread count.

For lower thread count and dataset, OpenMP performs better than p-thread, but when the dataset in increased, the overhead due to parallelization becomes a hurdle to the performance. From the figure, there is not much different in the performance between p-thread and OpenMP with fewer data points, whereas p-thread performs is better when the thread count (10) and the dataset is more (10000) is high.

8. Conclusion

Both the parallel code, p-thread, and OpenMP always performs better when compared to the serial code. Time increases with an increase in the number of centroids/clusters. Performance of OpenMP is better when dataset and thread count is low, whereas the Performance of P-thread is better when dataset and thread count is high. Resultant centroid might differ at every instance since they are initialized randomly. In the future, a parallel approach for other machine learning algorithm like Neural Networks will be explored and implemented with p-thread, OpenMP along with MPI and CUDA.

9. References

[1] www.kaggle.com - Credit Card Transaction Dataset

[2] Pattern Recognition and Machine Learning,
Christopher M. Bishop, Springer-Verlag ©2006 ISBN:0387310738

[3] Parallel k/h-Means Clustering for Large Data Sets
Kilian Stoel and Abdelkader Belkoniene